Nima Amir Dastmalchi
505320372
2022-10-05

# CS 131 Homework 1

## Hello Haskell

```
nimadastmalchi@Nimas-MacBook-Pro hw1 % ghci
GHCi, version 9.2.4: https://www.haskell.org/ghc/  :? for help
ghci> :load hello.hs
[1 of 1] Compiling Main             ( hello.hs, interpreted )
Ok, one module loaded.
ghci> length greeting
13
ghci>
```

## Installing Python 3

```
nimadastmalchi@Nimas-MacBook-Pro hw1 % python3
Python 3.9.12 (main, Mar 26 2022, 15:44:31)
[Clang 13.1.6 (clang-1316.0.21.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello world!")
Hello world!
>>>
```

## Problem 1

```haskell
-- Problem 1
largest :: String -> String -> String
largest str1 str2 = if (length str1) >= (length str2)
                        then str1
                        else str2
```

## Problem 2

Haskell evaluates functions from left to right. Thus, the expression `"reflect num+1"` is equivalent to the expression `"(reflect num) + 1"`. Note that `reflect num` results in an infinite recursion (and stack overflow) because the same function with the same argument is called on every recursive call (unless `num` is 0 and the base case is hit). Adding parenthesis around the addition/subtraction expressions will fix this problem:

```haskell
-- Problem 2
reflect :: Integer -> Integer
reflect 0 = 0
reflect num
   | num < 0 = (-1) + reflect (num+1)
   | num > 0 = 1 + reflect (num-1)
```

**Problem 3**

Part a)

Let `all_factors num` be the list of integers `x` such that `(mod num x) == 0` (i.e., `num` is divisible by `x`).

```haskell
-- Problem 3a
all_factors :: Integer -> [Integer]
all_factors num = [x | x <- [1..num], (mod num x) == 0]
```

Part b)

Let `perfect_numbers` be the list of integers `x` such that the sum of all factors of `x`, excluding itself, is equal to `x`.

```haskell
-- Problem 3b
perfect_numbers :: [Integer]
perfect_numbers = [x | x <- [1..], x == (sum $ init $ all_factors x)]
```

**Problem 4**

Assuming that the argument is always non-negative.

```haskell
-- Problem 4
-- version 1 - if/else statement
is_even :: Integer -> Bool
is_even x = if x == 0
              then True
              else is_odd (x - 1)

is_odd :: Integer -> Bool
is_odd x = if x == 0
             then False
             else is_even (x - 1)

-- version 2 - guards
is_even' :: Integer -> Bool
is_even' x
   | x == 0    = True
   | otherwise = is_odd' (x - 1)

is_odd' :: Integer -> Bool
is_odd' x
   | x == 0    = False
   | otherwise = is_even' (x - 1)
```

```haskell
-- version 3 - pattern matching
is_even'' :: Integer -> Bool
is_even'' 0 = True
is_even'' x = is_odd'' (x - 1)

is_odd'' :: Integer -> Bool
is_odd'' 0 = False
is_odd'' x = is_even'' (x - 1)
```

**Problem 5**

The first 2 patterns are the base cases. The pattern "**`count_occurrences (x : xs) (y : ys)`**" is the recursive case. In this case, if **`x == y`**, we can either count **`x`** as an element in the potential occurrence or ignore it and continue matching the rest of the lists. We make 2 recursive calls to account for these cases. Otherwise, if **`x != y`**, then we have no choice but to continue searching the rest of the second list for a match while retaining the **`x`** in the first list.

```haskell
-- Problem 5
count_occurrences :: (Eq a) => [a] -> [a] -> Integer
count_occurrences [] _ = 1
count_occurrences _ [] = 0
count_occurrences (x : xs) (y : ys)
   | x == y    = (count_occurrences xs ys) + (count_occurrences (x : xs) ys)
   | otherwise = count_occurrences (x : xs) ys
```